UNIVERSITAT POLITÈCNICA DE CATALUNYA
BARCELONATECH
Escola d'Enginyeria de Barcelona Est

TRABAJO FIN DE GRADO

Grado en Ingeniería Electrónica Industrial y Automática

APLICACIÓN ANDROID PARA VEHÍCULOS PESADOS

Memoria y Anexos

Autor: Pedro Sorriguieta Torre

Director: José Matas

Co-Director: Manuel Moreno-Eguilaz

Convocatoria: Octubre 2017


UNIVERSITAT POLITÈCNICA DE CATALUNYA
BARCELONATECH
UPC
Escola d'Enginyeria de Barcelona Est

Resumen

La tecnología forma parte de nuestro día a día, no solo en el trabajo, sino también en nuestra vida personal: Usamos nuestra tablet a primera hora para ver las noticias, tomamos café en nuestra cafetera último modelo, vamos a trabajar en un coche que poco tiene que envidiar al protagonista de El coche fantástico y podríamos seguir analizando nuestras acciones a lo largo de todo el día y nos daríamos cuenta de que prácticamente no hay ninguna tarea que realicemos, que no esté apoyada por una tecnología.

Las aplicaciones —también llamadas apps— están presentes en los teléfonos desde hace tiempo; de hecho, ya estaban incluidas en los sistemas operativos de Nokia o Blackberry años atrás.

Desde su irrupción en los años 60 en el mundo del automóvil con la sustitución del ruptor por el encendido transistorizado, la electrónica ha jugado un papel muy importante en este sector. Tanto, que a día de hoy podemos hablar de automóviles que buscan aparcamiento sin ayuda humana.

Las aplicaciones integran cada vez más funciones del vehículo, y, tras su implantación en el automóvil y el desarrollo de los wearables, los coches conectados están llamados a ser el siguiente paso en la evolución del llamado Internet de las Cosas.

El bus CAN es un protocolo de comunicaciones utilizado en los vehículos para la comunicación entre sus componentes electrónicos.

El objetivo del presente trabajo es el de integrar distintas funciones útiles para un conductor en una misma aplicación diseñada para sistemas Android. Nos permitirá visualizar el estado del vehículo (velocidad, volumen de la radio…) y ofrecer soluciones

inmediatas para resolución de problemas, como por ejemplo un nivel bajo de combustible. La aplicación ofrecerá además una interfaz que nos muestra nuestra localización en el mapa y distintos botones para ofrecernos lugares para parar a comer o dormir en un ratio alrededor nuestro.

Esta aplicación sólo tendrá utilidad en los vehículos pesados, cuyo lenguaje bus CAN ha sido unificado según el estándar J1939. En otro tipo de vehículo nos sería imposible decodificar los datos obtenidos del bus CAN por la impermeabilidad que presentan los fabricantes de turismos respecto a los lenguajes utilizados en sus vehículos.

UNIVERSITAT POLITÈCNICA DE CATALUNYA
BARCELONATECH
Escola d'Enginyeria de Barcelona Est

ÍNDICE

Contenido

1. ACRÓNIMOS 3

2. INTRODUCCIÓN 7

2.1 ORIGEN DEL PROYETO 7

2.2 ACERCA DEL PROYECTO 9

3. INVESTIGACIÓN PREVIA 12

3.1. FUNDAMENTOS DEL BUS CAN 12

3.2. INTERFAZ USB-BUS CAN 26

3.3. ESTÁNDAR J1939 30

4. IMPLEMENTACIÓN SOFTWARE 35

4.1. ELECCIÓN LENGUAJE DE PROGRAMACIÓN 35

4.1.1. Candidatos 35

4.1.2. Elección final: B4A 36

4.2. BIBLIOTECA USB-CAN 37

4.2.1. Class Globals 38

4.2.2. Public Subs 39

4.3 DESARROLLO DE LA APLICACIÓN 50

4.3.1. Primer problema: integrar Google Services a la Aplicación 50

4.3.2. Pantalla principal 54

4.3.4. Panel 2 55

4.3.5. Panel 3 55

5. IMPACTO MEDIOAMBIENTAL 59

BIBLIOGRAFÍA. 67

1. ACRÓNIMOS

1. ACRÓNIMOS

CAN	Controller Area Network
CSMA/CD	Carrer Sense Multiple Access with Colision Detection
DA	Dirección de destino
ECU	Electronic Control Unit
EMIs	Interferencias electormagnéticas
GE	Grupo de extensión
IDE	Integrated Development Environment
LLC	Control de enlace lógico
OTG	On the go
PDU	Protocol Data Unit
PNG	Parameter Group Number
USB	Universal Serial Bus
VB	Visual Basic

2. INTRODUCCIÓN

2. INTRODUCCIÓN

2.1 ORIGEN DEL PROYETO

Con el curso de los años el mundo de las tecnologías móviles y el uso de dispositivos tales como teléfonos inteligentes o smartphones y tabletas electrónicas ha explotado. Este hecho ha provocado que se genere un nuevo mercado lleno de oportunidades de negocio. Por ejemplo, se podría hacer referencia al negocio que se genera con las redes sociales y la publicidad asociada, así como también en las aplicaciones de entretenimiento. Estas tecnolodías nos permiten comunicarnos con otras personas de forma instántanea, consultar información desde cualquier lugar con un poco de cobertura o realizar compras con nuestro teléfono móvil. Estos serían un ejemplo del inmenso mundo de oportunidades que nos brindan las aplicaciones móviles.

Diseñar aplicaciones no es una tarea fácil, especialmente si no has trabajado antes con ellas o si vienes del mundo web, un contexto radicalmente diferente al que ahora nos enfrentamos. Para los diseñadores es todo un desafío empezar a diseñar para móviles, pero también, es una buena oportunidad para meterse en este ámbito donde los clientes demandan cada vez más y mejores herramientas de comunicación y promoción.

La creación de aplicaciones implica la dificultad de aprender un nuevo lenguaje de programación y diseño gráfico, además del atractivo de diseñar un proyecto desde cero. Desarrollar una idea, estructurarla y llevarla a cabo. Es esto lo que motivó la decisión de realizar el proyecto sobre el diseño de una aplicación Android.

El mundo del automóvil está cada vez más estrechamente relacionado con las aplicaciones para Smartphone o iPhone. Decidimos en nuestro caso desarrollar la aplicación en Android por su más amplio mercado. Nuestro programa está destinado a conductores de vehículos pesados (camiones), clientes generalmente con un nivel

adquisitivo más humilde comparado con el sector de mercado al que se destinan los p

roductos de Apple.

La industria del automóvil, desde hace unos años hasta ahora, está incorporando a sus vehículos cada vez más aplicaciones y sensores, cuyo objetivo es, principalmente, el de aumentar la seguridad de sus pasajeros. Sin ir más lejos, ya hay en el mercado un gran número de vehículos que tienen incorporados varios sensores capaces de detectar obstáculos en la carretera y aminorar la velocidad para evitar el impacto. Si esto hasta hace pocos años era casi impensable, la pregunta es, ¿hasta dónde podrá llegar esta tecnología?

No podemos responder a la respuesta anterior. Sin embargo, lo que sí sabemos, es entorno a qué se va a trabajar estos próximos años, para que nuestros coches sean cada vez más seguros, y no es otra cosa que la comunicación coche a coche. Ya en este año han comenzado las primeras pruebas reales en las cuales los coches se comunicaban entre sí, intercambiando información relativa a su posición, velocidad, trayectoria... entre otras muchas cuestiones, y todo ello con el objetivo de reducir al máximo el número de accidentes. Desconocemo cuándo tendremos en el mercado coches con esta tecnología incorporada, pero sí sabemos es que, tarde o temprano, llegarán.

Este boom de mercado y el atractivo del mundo del automóvil para un ingeniero han supuesto la decisión de desarrollar nuestra aplicación enfocada a este sector.

Desde la escuela de la UPC ya se han venido desarrollando proyectos similares que han facilitado la decisión y desarrollo de este proyecto.

2.2 ACERCA DEL PROYECTO

Objetivos

Como ya se ha comentado antes, el objetivo principal de este proyecto es desarrollar una aplicación en formato Android que tenga su utilidad para el sector de mercado de los conductores de vehículos pesados.

Para ello el objetivo es integrar varias funcionalidades y accesorios en dicha aplicación. Éstos accesorios y funcionalidades serían:

a) Mapa con localización en tiempo real.

Tendrá además un botón de búsqueda con el que podremos encontrar sitios de interés y localizarnos en el mapa, con la función añadida de ofrecernos un botón que nos permita "navegar" hasta nuestro destino con la aplicación Navigator de Google Maps.

b) Búsqueda de establecimientos para comer.

La aplicación constará con un panel (Los paneles se utilizan, entre otras cosas, para situar un conjunto de controles que pueden aparecer y desaparecer, desplazarse, anularse...) que nos ayudará a buscar sitios cercanos donde poder parar a comer o elegir en el propio mapa el que más convenga al usuario.

A este panel se podrá acceder desde la pantalla principal mediante un botón.

c) Búsqueda de establecimientos para dormir.

De la misma manera que el panel expuesto en el apartado anterior nos ayudará a buscar lugares para dormir este panel nos mostrará los lugares más cercanos en los que el conductor podrá parar a dormir o una búsqueda personalizada.

Asimismo, este panel también podrá ser abierto desde la pantalla principal mediante otro botón.

d) Visualización estado del coche.

En otro panel abierto mediante un botón en la pantalla principal, se podrán visualizar las distintas lecturas obtenidas mediante el bus CAN. Esta pantalla incluye un botón de resolución de problemas, dirigiéndonos al lugar más cercano que nos solucione la incidencia.

Alcance del proyecto

En un principio, esta aplicación podría estar destinada a un sector mucho más amplio del que finalmente se ha abarcado. El objetivo de la aplicación es que sea básica e intuitiva para facilitar el uso mientras se conduce, además de que no presente ninguna dificultad a los usuarios principiantes. Esto permitiría que cualquier conductor con conocimientos básicos en Smartphone podría manejar nuestra aplicación sin dificultad.

Sin embargo, existe un inconveniente:

Es muy sencillo conectar nuestro Smartphone al bus CAN de nuestro coche, pero eso no quiere decir que podamos descifrar o interpretar fácilmente los mensajes que leemos. La codificación de estos mensajes depende de cada fabricante y están generalmente bajo un manto de confidencialidad inaccesible para nosotros.

Es así que utilizaremos el estándar J1939, presente por norma en vehículos pesados, para tratar de abarcar el sector de mercado más amplio a nuestro alcance.

3. Investigación previa

3. INVESTIGACIÓN PREVIA

En este capítulo se presenta una introducción teórica al proyecto, donde estudiaremos el funcionamiento del bus CAN, el interfaz CAN/USB y se realiza una pequeña introducción al estándar J1939.

3.1. FUNDAMENTOS DEL BUS CAN

Esta información nos es clave porque será la que nos permitirá comprender el funcionamiento del bus, cómo se comunica y la manera de tratar los datos.

A continuación se desarrolla una corta explicación del bus CAN, más concretamente se detallan los aspectos que conciernen a este proyecto. Esta información ha sido extraída de la amplia bibliografía que existe sobre este tema referenciada al final del trabajo y que el lector puede consultar si desea ampliar los con conocimientos sobre dicho tema

Razones para el origen del CAN

El incremento del numero de dispositvos Electrónicos en los Automóviles y el aumento de las necesidades de cableado y su complejidad propiciaron la posibilidad de conectar todos los dispositivos a un bus fiable, robusto y con alta inmunidad al ruido entre otras ventajas. Además, este bus debía permitir altas velocidades de transmisión en entornos difíciles debido a los cambios de temperatura, vibraciones e interferencias que puede sufrir un automóvil en uso. Por estos motivos surgió la necesidad de crear un bus que tiene también aplicaciones Industriales en sectores distintos al automovilístico.

Historia

El bus CAN (Controller Area Network), fue desarrollado por la compañía Robert Bosch en 1982, posibilita la comunicación entre los diferentes dispositivos electrónicos que existen en un vehículo. Inicialmente se pensó en este instrumento como bus de campo pero realmente encontró utilidad en el sector del automóvil para interconectar el bus

de confort y seguridad entre otros. Diez años después de su creación se lanzó el Mercedes Clase E que fue el primer coche en incorporar el bus CAN, en ese momento (1992) fue diseñado para permitir la comunicación fiable entre centralitas electrónicas basadas en micoprocesador, ECUs ("Electronic Control Unit") y reducir cableado. Existen diferentes versiones del Bus CAN que atienden a diferentes especificaciones. En Europa el bus CAN se ha convertido en un estándar "de facto" con carácter internacional por la norma ISO 11898, y para aplicaciones en automoción por la SAE J22584 (turismos) y SAE J1939 (camiones y autobuses). Las ventajas que aportó su creación fueron una reducción de costes y una mejora de la flexibilidad entre otras.

El bus CAN es un protocolo de serie asíncrono del tipo CSMA/CD ("Carrier Sense Mul:ple Access with Collision Detection"). Se trata de un medio compartido (multiplexado) que sigue el protocolo "Multicast", es decir, todo el mundo puede hablar (de uno en uno) y escuchar. Que sea del tipo "CSMA" implica que cada nodod de la red debe monitoritzar el bus y en caso de detectar actividad nula, puede alertar con un mensaje. En referencia al tipo "CD" conlleva que si hay dos nodos de la red trasmitiendo un mensaje, ambos detectan la colisión, dicho conflicto se resuelve con un método de arbitración basado en prioridades. Se utilizan un par de cables trenzados (bus diferencial) para conseguir alta inmunidad a las interferencias electromagnéticas (EMIs). La impedancia caracteristica de esta linea es del orden de 120Ω por lo que se emplean impedancias (resistencias) de este valor para en ambos extremos del bus para evitar ondas reflejadas y que el bus se convierta en una antena. Si longitud màxima es de 1000m (a 40Kbps). Y su velocidad de 1Mbps (con una longitud de 40m). En los coches se utiliza a 125 kbit/s y a 500 kbit/s.

Estructura: Modelo de Capas en el bus CAN.

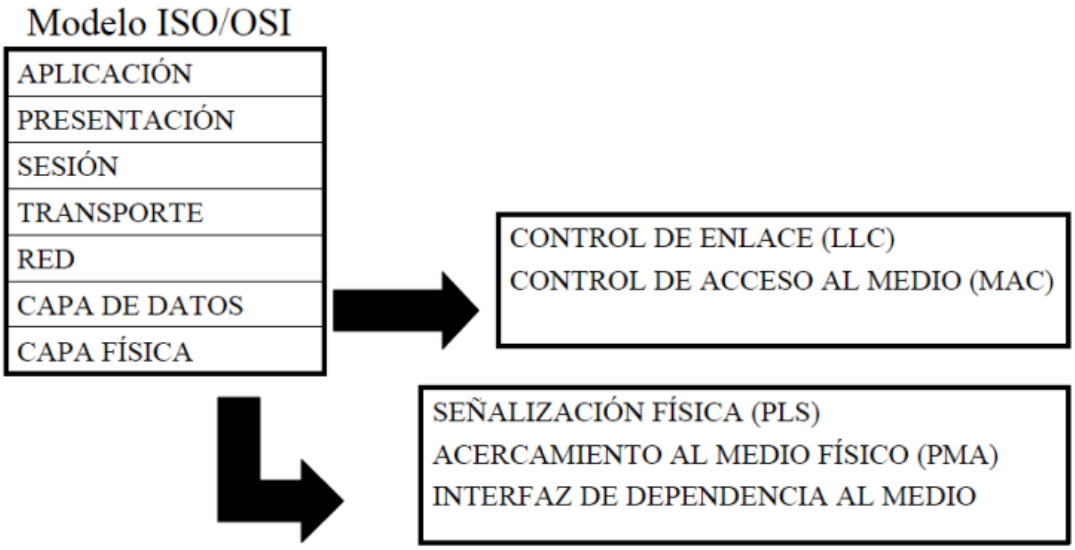


Figura 1.-Esquema del modelo a capas del bus CAN.

El modelo de capas del bus CAN se puede desglosar en dos partes:

-CAPA DE ENLACE DE DATOS:

En esta tienen lugar dos tipos de control. Por una parte, el control de enlace lógico (LLC), que se encarga de los filtros de los mensajes y proporciona Servicios durante la tranferencia y petición de datos. Además decidí que mensajes recibidos de MAC se aceptan. También proporciona medios para el restablecimiento y notificación de sobrecargas del bus. Por otra parte, el control de acceso al medio (MAC) que representa el núcleo del protocolo CAN, presentando los mensajes recibidos a la subcapa LLC y acepta cuales de ellos son transmitidos a dicha subcapa, actuando como filtro, es decir, es responsable de la trama de mensajes, arbitraje, reconocimiento, detección de error y señalización. En esta subcapa se decide que si el bus está libre para comenzar una nueva transmisión o si la recepción acaba de comenzar.

-CAPA FÍSICA:

Es la encargada de definir que señales se transmiten y de tratar la descripción del bit de cronometraje, la codificación de bit y la sincronización.

Propiedades

-Priorización del mensaje

-Garantía de los tiempos de retardo

-Flexibilidad de la configuración

-Recepción múltiple con tiempos de sincronización

- Robustez en sistemas de amplios datos

- Multimaestro

-Detección de error y señalización

-Retransmisión automática de mensajes corruptos tan pronto como el bus está libre de nuevo.

-Distinción entre errores temporales y fallos permanentes de nodos.

-Desconexión automática de nodos defectuosos.

Conceptos básicos:

-Estructura de los mensajes CAN:

El esquema sencillo de la trama de un mensaje CAN cuando se envían datos es la siguiente:

- Identificador (11 bits, extendido: 29 bits): Indica el tipo de mensaje que se está enviando. Además sirve para asignar la prioridad del mensaje. Cuando menor es el identificador, mayor es la prioridad del mensaje.

- DLC (4 bits): *Data Length Code*. Indica el número de bytes de datos que se van a transmitir en el mensaje. Puede valer entre 0 y 8.

- Datos (hasta 64 bits, 8 bytes como máximo): Mensaje que se envía..

Además de los campos anteriores también se incluyen en la estructura campos adicionales que no aportan información pero son necesarios para detectar el inicio y el fin del mensaje así como para detectar errores. En la Figura X se muestra el esquema de un mensaje CAN con todos los campos anteriores:

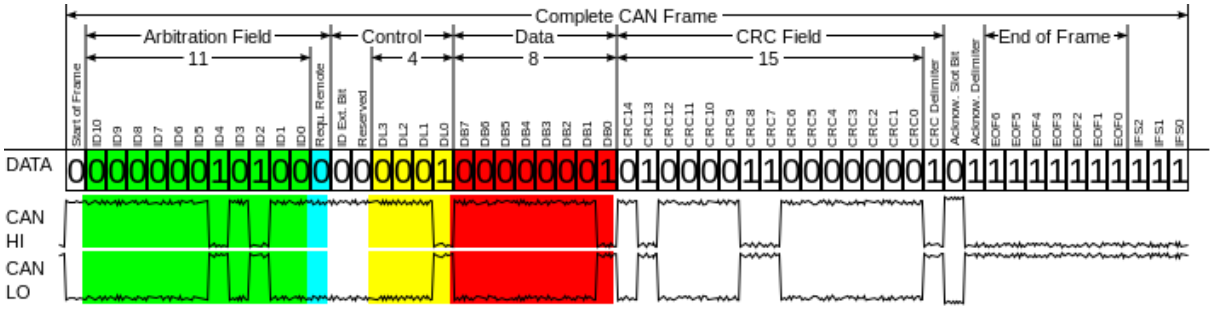


Figura 2. Estructura de un mensaje CAN. Se muestran los campos de Identificador en verde, DLC en amarillo y de Datos en rojo.

En cuanto a la interpretación de mensajes CAN que se localiza en la capa de aplicaciones, que define como se codifican los datos contenidos dentro de la estructura de un mensaje CAN descrita anteriormente. El problema surge si se desconocen cuáles son los

identificadores que representan la información a la cual queremos acceder y tampoco se conoce como están codificados los datos. El estándar del bus CAN define la estructura del paquete donde viajan los datos, pero no especifica una manera estándar de codificarlos, luego cada fabricante puede establecer su propia forma de codificar los datos (llamado capa de protocolo de alto nivel). Algunos sí que se pueden conocer debido a que son necesarios para realizar diagnóstico, pero en principio la política del fabricante es la de la ocultación (en principio esto se justifica para no revelar secretos a la competencia pero otras veces el motivo es para evitar prácticas irregulares).

En referencia la velocidad de transmisión hay que destacar que es irregular ya que puede ser diferente dependiendo del sistemsa, la única condición es que debe ser uniforme en un sistema. La prioridad del mensaje es definida por el identificador.

Multimaestro

El hecho de que el bus sea multimaestro implica que si el bus está libre, cualquier nodo puede trasmitir un mensaje. Como se ha introducido anteriorment, si dos nodos empiezan a transmitir simultáneamente el conflicto de acceso al bus es resueslto por arbitraje con el uso del identificador. Este mecanismo garantiza que ni la información ni el tiempo se pierdan. Cuando una trama de datos y una trama remota se inician al mismo tiempo prevalece la primera. Durante el arbitraje todos los transmissores comparan el nivel del bit transmitido con el nivel del bus. Si los niveles son iguales, la unidad puede enviar, si son distintos se pierde el arbitraje y la unidad debe retirarsee sin enviar otro bit.

Seguridad

En todos los nodos CAN se implementan medidas especiales para la detección de errores, señalización y auto-chequeo. a detección de error se da por monitorización (comparación de niveles de bit), por CRC, chequeo de la trama del mensaje o porBit Stuffing.

La señalización de error y tiempo de restablecimiento se da cuando los mensajes corruptos son reconocidos por cualquier nodo. Entonces estos mensajes se abortan y se transfieren automàticament. El tiempo de restablecimiento desde la detección del error es de 31 bits. Los nodos CAN ditinguen tanto perturbaciones cortas como fallos permanentes y estos nodos por defecto se desconectan. Las conexiones no tienen límite teórico y en la pràctica el número total de unidades estará limitado por el tiempo y las cargas eléctrucas. El bus se componen por un único canal de transmisión, la forma de implementación no se fija en las especializaciones (único hilo, dos hilos diferenciales, fibra óptica, etc). El bus también puede adquirir los valores de dominante o recesivo y siempre todos los mensajes pasan por una etapa de reconocimiento en la que los receptores comprueban su fiabilidad. Además incluye un modo de ahorro de consumo de potencia (modo sleep) que se combina con el modo wake-up.

Protocolo CAN

El protocolo Can està basado en mensajes y por tanto, no tiene direccionamiento de nodo a nodo. La priorización y el direccionamento esta contenido en los datos transmitidos, y esta información es recibida por todos los nodos del sistema, después cada nodo decidí si el mensaje o trama debe ser aceptado o descartado. Un único mensaje puede destinarse para un nodo en particular o para varios y un nodo tiene la habilidad de pedir información de otros nodos (Remote Transmit Request). Los nodos pueden añadirse sin necesidad de cambios en el sistema.

En el caso de vehículos pesados (camiones y buses) sí que existe un estándar definido en el estándar SAE J1939 para codificar la información que nos permite interpretar la información recibida. Esto es así porque se debe asegurar la compatibilidad entre la cabeza tractora y el remolque, en el caso de que no correspondan al mismo fabricante. En este caso se especifica que el identificador tiene un formato conocido en formato extendido (29 bits), siendo la velocidad de transmisión de 250 Kbps.

El estándar define un formato para la parte de identificador de un mensaje CAN. El identificador siempre incluirá un *Parameter Group Number* (PGN). Este PGN identifica de manera unívoca un grupo de parámetros. Estos grupos de parámetros (definidos en las especificaciones SAE J1939-71) combinan señales similares o asociadas.

A su vez, el PGN es una combinación de los bits de *extended Data Page* (reservado, siempre 0), el bit de *Data Page* y el PDU (*Protocol Data Unit*) Format i el PDU Specific (*Destination Address/Group Extension*).

Se pueden formar dos tipos de PGN:

a) Si PDU < 240 (peer-to-peer): PDU Specific contiene la dirección de destino (DA). Una dirección global (255) también puede utilizarse como dirección de destino.

b) Si PDU ≥ 240 (difusión): el formato PDU junto con la Extensión de Grupo (GE) en el PDU específico forma el campo del PGN del grupo de parámetros que se quieren transmitir .

Finalemente, los últimos 8 bits del identificador contienen la dirección del dispositivo que transmite el mensaje. La dirección es la etiqueta o handle, que se asigna para proporcionar una forma de acceder de forma exclusiva un dispositivo en la red. Para una red dada cada dirección debe ser única (254 disponibles). Esto significa que dos dispositivos diferentes (ECU) no pueden utilizar la misma dirección.

Tramas CAN

El protocolo CAN define cuatro tipos de mensajes:

– Tramas de datos (Data Frame) que transmite información de un nodo a cualquiera de los restantes. Esta trama esta formada por campos que porporcionan información addicional sobre los mensajes definidos en CAN y que se detallaran a continuación.

– Trama remota (Remote Frame) se trata de una trama de datos con el bit RTR=1.

– Tramas de error son aquelles generadas por nodos que detectan cualquiera de los errores de protocolo definidos por CAN

– Tramas de overload que son generadas por nodos que necesitan más tiempo para procesar los mensajes ya recibidos

Campos adicionales de las trames de datos

• Campo de arbitraje: Se utiliza para priorizar los mensajes en el bus. Está formado por 12 o 32 bits. Puede ser una trama estándar: 11 bits de identificación y un bit RTR o trama extendida: 29 bits de identificación, 1 bit para definir el mensaje como trama extendida, un bit SRR no usado, un bit RTR.

• Campo de control. Formado por seis bits, el bit IDE de mayor peso y el bit RBO reservado. Los cuatro de menor peso definen la longitud de los datos (DLC). Campo de datos. Nº de bytes determinado por DLC. Los R no tienen campo de datos.

• Campo de CRC formados por 15 bits y un delimitador CRC, es utilizado por receptores para detección de errores de transmisión

• Campo de confirmación (ACK). El nodo receptor indica recepción correcta del mensaje, poniendo un bit dominante en el flag ACK de la trama.

A continuación se adjuntan las estructuras de algunas tramas.

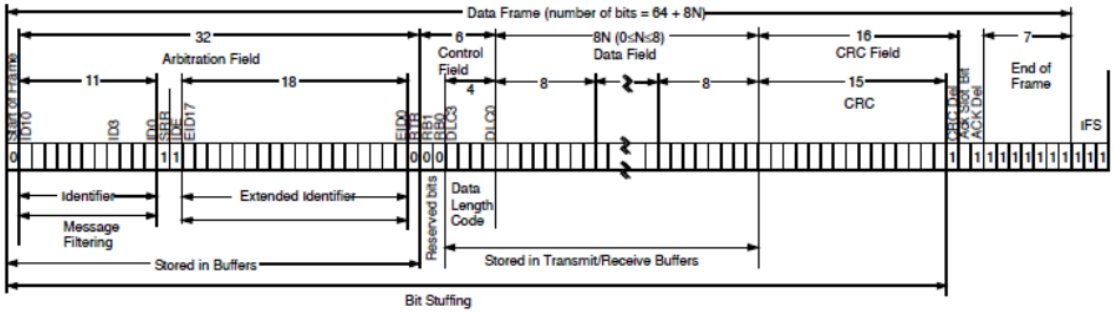


Figura 3.- Trama de datos estendida.

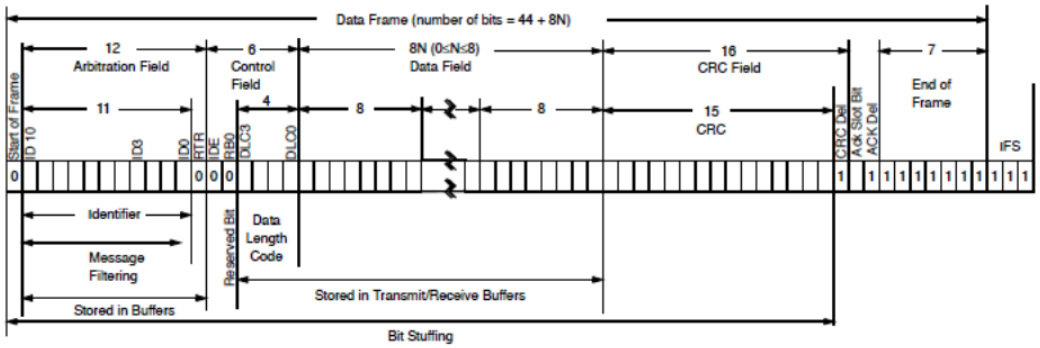


Figura 4.- Trama de datos estándar.

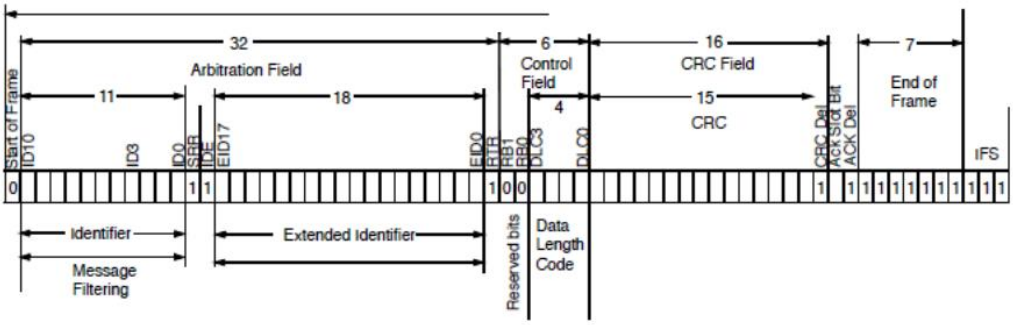


Figura 5.- Trama de datos remota.

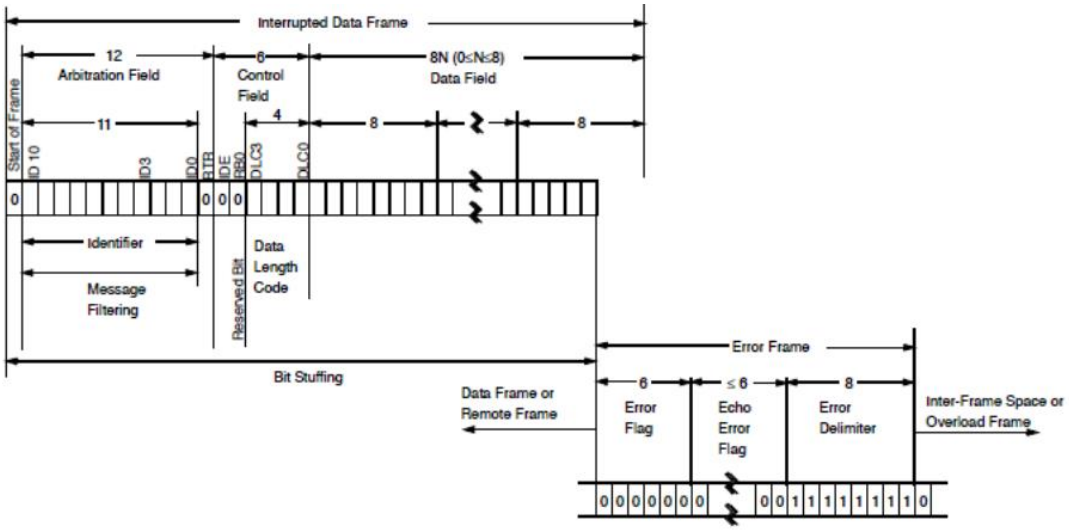


Figura 6. Trama de error.

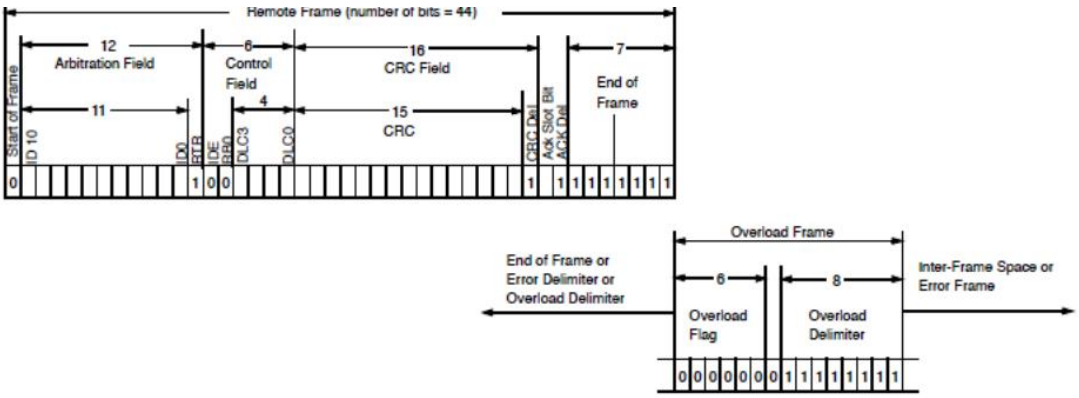


Figura 7.-Trama overload.

Tipos de detección de errores

-Error CRC. Si al menos un nodo no recibe correctamente el mensaje, éste genera la trama de error y el mensaje es reenviado.

-Error de ACK. El nodo transmisor comprueba si el flag ACK, enviado como recesivo, contiene un bit dominante. Este bit dominante reconocerá que al menos un nodo ha recibido correctamente el mensaje.

-Error de forma. Si cualquier nodo detecta un bit dominante en uno de los cuatro segmentos del mensaje: Final de trama, espacio entre tramas, delimitador ACK o delimitador CRC, el protocolo CAN define esto como una violación de la forma.

-Error de bit. Si un trasmisor envía un bit dominante y detecta un bit recesivo (o viceversa) cuando monitorea el nivel del bus actual y lo compara con el bit enviado. Se excluye el bit ACK y el arbitraje.

-Error de stuff. Los nodos receptores se sincronizan con la transición. Si hay más de 5 bits de la misma polaridad, CAN pone un bit de polaridad opuesta (stuffing bit). Si se detectan 6 bits con la misma polaridad se produce un error de stuff.

Esquema de implementación del sistema con el controlador MCP2510.

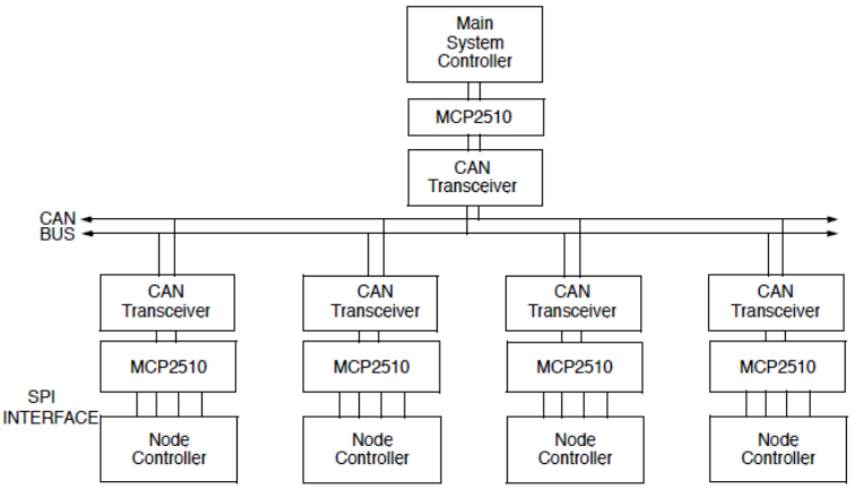


Figura 8.- Esquema controlador MCP2510.

Las características de este controlador se pueden dividir en:

-Motor de protocolo CAN.

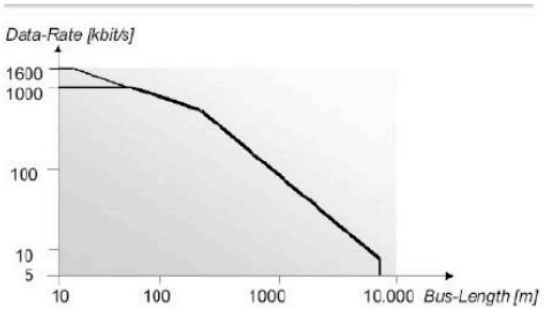
-Control lógico.

-Protocolo SPI.

-Pines de interrupción (uno gerenal más dos de los registros receptores (opcional).

-Pines de inicio de transmisión inmediata (opcional).

-Velocidad de transmisión versus distancia cable:



Bit Rate	Bus Length	Nominal Bit-Time
1 Mbit/s	30 m	1 µs
800 kbit/s	50 m	1,25 µs
500 kbit/s	100 m	2 µs
250 kbit/s	250 m	4 µs
125 kbit/s	500 m	8 µs
62,5 kbit/s	1000 m	20 µs
20 kbit/s	2500 m	50 µs
10 kbit/s	5000 m	100 µs

Figura 9. Variación de la velocidad de transmisión en función de la distancia de cable.

Características eléctricas

-El cable debe terminarse en ambos extremos por la impedància de carga de Ohm.

-El controlador CAN se conecta con el Transceiver vía serie.

-El nodo detecta una condición recesiva del bus cuando el voltaje de CAN_H no es más alto que el voltaje de CAN_L mas 0.5 V.

-Si el voltaje de CAN_H es al menos 0.9 V más alto que CAN_L, se detectará una condición dominante.

-El voltaje nominal en el estado dominante es CAN_H=3.5V y CAN_L=1.5V.

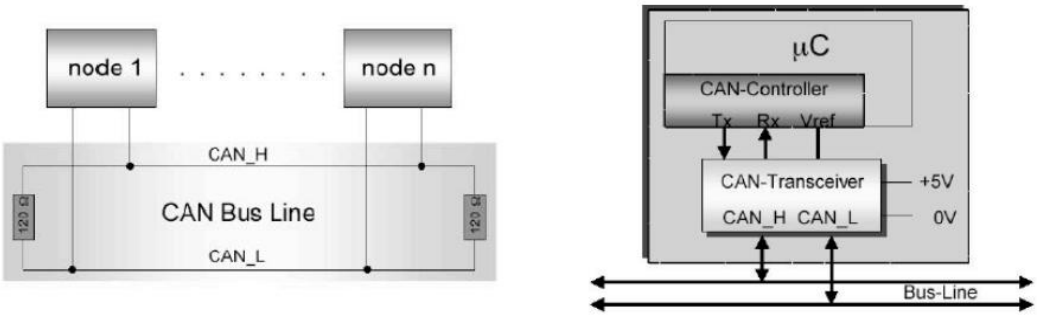


Figura 10.- Aspecto del bus y de cada nodo respectivamente.

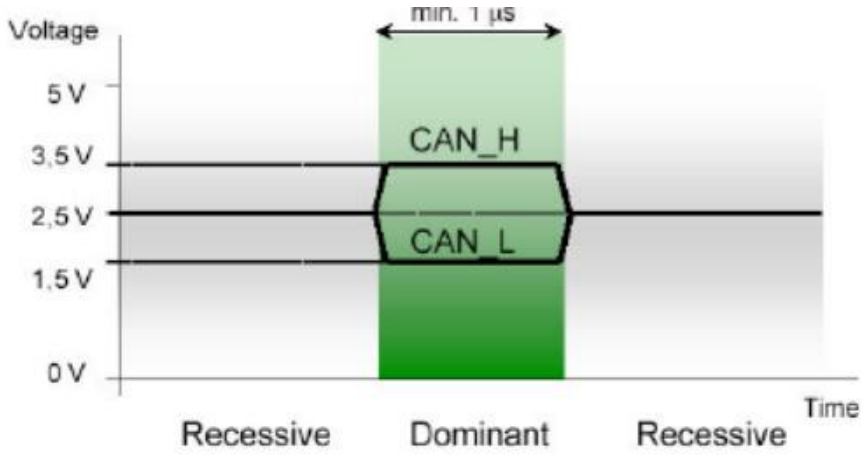


Figura 11.- Niveles de tensión nominales

3.2. INTERFAZ USB-BUS CAN

Este trabajo se basa en conectar un smartphone con un puerto USB (Universal Serial Bus) a un bus CAN. Por este hecho se debe disponer de un adaptador o interfaz que capte la señal transmitida por el par de cables trenzados como si fuese un nodo de la red y la convierta en una información legible por un puerto USB de un ordenador o tableta.

Existen bastantes interfaces CAN/USB Comerciales como se puede comprovar visitando el siguiente enlace, Anonymous Contributors, "CAN Interface Collection," *CAN Wiki*, 2015. [Online]. Available: http://www.can-wiki.info/doku.php?id=can_interfaces:main. Por ejemplo, Kvaser es una empresa que fabrica este tipo de dispositivos, que además requieren de un software (por ejemplo CANalyzer de Vector Informatik) y un PC para hacer el tratamiento de los datos. Pero en este caso el precio es altoo, y además el software no funciona con todos los dispositivos. Por tanto, en nuestro caso utilizaremos el interfaz de bajo coste USBtin y desarrollaremos nuestro propio software.

Como acabamos de mencionar en nuestro caso hemos usado la tarjeta USBtin. Esta es una tarjeta de barata adquisición creada por Thomas Fischl a partir de un proyecto con filosofía *Open Source*, por lo que se dispone de mucha información en la red, si al lector le interesa saber más acerca del tema puede visitar el siguiente enlace, http://www.fischl.de/usbtin/ . La targeta tiene un precio reducido (alrededor de 40 euros) y cubre todas nuestras necesidades.

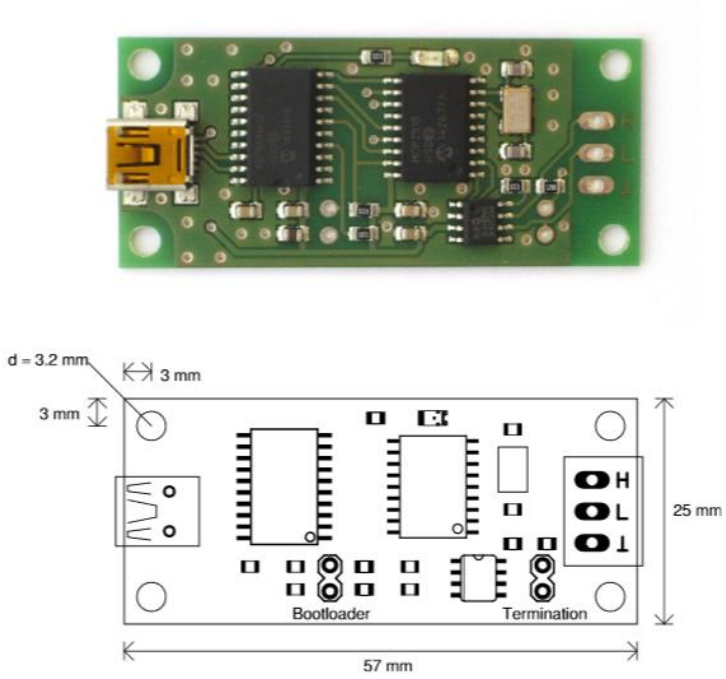


Figura 12.- Esquema y apariencia de la targeta USBtin.

Esta tarjeta se puede conectar a un PC con sistema operativo Windows o Linux mediante un software que se puede encontrar desarrollado en la web. También hay que advertir que se trata de un producto no destinado a aplicaciones comerciales y que por tanto no cumple con las regulaciones establecidas para dicho objetivo como la CE entre otras.

A nivel físico, debemos comentar que la tajeta dispone de tres pines correspondientes al bus CAN (CAN_High, CAN_Low y Masa) en un lateral y un conector USB hembra en el otro.

Esta tarjeta se controla a traves del puerto USB, al cual podemos enviar mensajes ASCII para enviar un mensaje CAN, o para recibirlos. La configuración de distintos de sus aspectos puede ser modelada por los usuarios. Además, hay que remarcar que el puerto USB de la tableta debe ser del tipo OTG (*On The Go*), esto significa que el smartphone puede comportarse como esclavo cuando se establece una conexión con la tarjeta interfaz.

En cuanto al control de esta tarjeta, esta se puede controlar, como hemos mencionado, enviando comandos en forma de cadenas de caracteres ASCII por el puerto USB. B4A cuenta con una librería que permite el control de puerto USB. Estas cadenas están definidas en los manuales de la tarjeta USBtin. Por ejemplo, para enviar un mensaje CAN, transmitimos por el puerto USB el comando siguiente:

tiiildd..[CR]	Transmit standard (11 bit) frame. Iii: Identifier in hexadecimal format (000-7FF) l: Data length (0-8) dd: Data byte value in hexadecimal format (00-FF)

Es decir si enviamos t001411223344[CR], estamos enviando un mensaje CAN en el que:

- id=001h
- dlc=4
- data=11 22 33 44

Todos los comandos funcionan de manera similar tanto para enviar como para recibir mensajes. Una lista exhaustiva de los comandos se incluye como anexo.

Otro aspecto a recalcar es la opción de activar el *timestamping*, es decir, añadir después del mensaje CAN el tiempo correspondiente al instante en que el mensaje CAN fue recibido por la tarjeta, que podría ser significativamente diferente al tiempo en que nuestro dispositivo electrónico reciba el mensaje por el puerto USB.

También hay que parar atención en las máscara y filtros que se pueden emprar. En un bus CAN todos los nodos tienen permiso para leer el tráfico de toda la red, pero cuando reciben un mensaje, estos nodos pueden realizar un test de aceptación para determinar si lo procesan o no (filtrado de mensajes).

USBtin permite configurarse para establecer máscaras que realicen el filtrado de mensajes de forma automática por hardware. La máscara se utiliza para indicar cuáles de los bits del identificador se utilizarán para realizar el proceso de filtrado.

Este filtrado lo realiza el propio hardware de la electrónica de la tarjeta y permite aligerar la carga de trabajo aguas abajo del filtro, al no procesarse los mensajes que no se van a necesitar.

Otro interfaz CAN/USB que se podria utilitzar es el Kvaser que permite la comunicación entre el puerto USB y un bus CAN. Se puede conectar a un PC y permite establecer comunicación con el bus CAN (mediante un conector tipo DSUB 9). El precio de este dispositivo es más elevado, en torno de los 300 euros.

Figura 13.- Interfaz Kvaser.

Lamentablemente USBtin solo acepta identificadores de 11 bits, mientras que nosotros necesitamos 29 bits (formato extendido), por lo que no se puede hacer uso de esta característica de la tarjeta. Puesto que se encuentra disponible en el Departamento de

Ingeniería Electrónica, se utilizará para probar nuestro sistema, emulando el comportamiento del nodo (ECU del camión) que envía la velocidad del vehículo al bus CAN. Para ello, se utilizará un código en Python basado en el paquete CANLIB, que contiene las librerías y drivers necesarios para trabajar en entorno Windows.

3.3. ESTÁNDAR J1939

Aspectos generales

¿Qué es el estándar estándar J1939?

Es un protocolo de nivel alto basado en Controller Area Network. Provee de comunicaciones en serie entre unidades de control electrónico en cualquier tipo de vehículo pesado.

Las características de este protocolo están basadas en los estándares J1708 (normativa RS485) y J1587.

Es un ingenioso protocolo desarrollado con muy poca sobrecarga de información. El papel principal lo ejercen los datos, no como en otros protocolos de alto nivel en los que un montón de funciones ejercen este papel.

Aprovecha todas las ventajas y puntos fuertes que nos pueden ofrecer las características del bus CAN.

¿En qué ámbitos es utilizado?

Podemos ver su aplicación en aplicaciones de diésel power-train, en la red interna de camiones y autobuses, vehículos agrónomos (ISO 11783), conexiones de los tráilers, vehículos militares (MiLCAN), sistemas de gestión de flotas, caravanas o en los sistemas de navegación de los marines (NMEA2000)

¿Qué ventajas aprovecha del bus CAN?

- Máxima fiabilidad.
- Excelente detección de errores y confinamiento de dicho error.
- Excelente manejo de las colisiones de mensajes en el bus.

Especificaciones J1939

- Par de cables trenzados protegido.
- Máxima longitud de la red de 40m.
- Velocidad de transmisión estándar de 250kBits por segundo.
- Usa una identificación de mensaje de 29 bits.
- Tiene un máximo de 30 nodos (ECUs) conectados.

Con esta introducción tenemos la base técnica para empezar a desarrollar la aplicación.

4. Implementación Software

4. IMPLEMENTACIÓN SOFTWARE

Este capítulo está dividido en tres apartados: en el primero discutiremos la elección del lenguaje a utilizar para la programación, en el segundo profundizamos en las funcionalidades de la biblioteca USBCAN, herramienta básica y fundamental de nuestra aplicación, y en el tercero finalmente se expone el desarrollo de nuestra app.

4.1. ELECCIÓN LENGUAJE DE PROGRAMACIÓN

Una vez fijados los objetivos del proyecto y la manera de llevarlo a cabo, uno de los puntos clave será el de elegir el lenguaje de programación a utilizar realizando una pequeña visión de sus ventajas y desventajas.

4.1.1. Candidatos

Barajamos tres opciones en un principio: Java (Android Studio), Python y Basic for Android (B4A):

1. Java (Android Studio)

Hablando de lenguajes de programación para aplicaciones Android, Java es el lenguaje utilizado por Android Studio, el entorno de desarrollo integrado (IDE) oficial en el que Android está basado –un IDE es una aplicación software que ayuda al usurio a implementar el software--. Ser la IDE oficial para Android siginifica que siempre tendrá soporte en internet y será constantemente actualizado. Por lo tanto, la variedad de aplicaciones que pueden ser desarrolladas será más amplia. Por otra parte, no es fácil para un principiante en el campo de la programación empezar de cero en un lenguaje como Java y poder observar resultados plausibles en poco tiempo. Esto siginifica que tiene una curva de aprendizaje muy lenta, que puede no ajustarse con los tiempos que manejamos.

2. Python (Kivy)

Kivy es una librería de Python desarrollada con la finalidad de diseñar apps para smartphone. Sus puntos fuertes serían su velocidad, ya sea en términos de desarrollo de la aplicación como en términos de ejecución, su flexibilidad para ser ejecutada en distintos tipos de dispositivo y su facilidad de lenguaje de programación. Pero su principal inconveniente, clave en nuestro proyecto, es la posibilidad de aparición de problemas cuando la aplicación accede a señales de entrantes o salientes. El departamento de Electrónica ya trabajó con este sistema y pudieron observarse este tipo de problemas con esta aplicación.

3. Basic for Android

El lenguaje de programación de Basic For Android es sencillo y simple. Está basado en el Visual-Basic (VB) IDE y en lenguaje de programación, es decir, consta de una ventana para un diseño visual fácil e intuitivo, y otra ventana principal en la que implementar el código. Es apto para principiantes por esta razón, y hay tutoriales que permitirán al usuario realizar su primera aplicación en unas pocas horas. Otro de los beneficios de B4A es su fórum online, con una comunidad de usuarios muy activa, con tutoriales y ejemplos al acceso del usuario. Su inconveniente es el precio, que sería alrededor de unos 60€.

4.1.2. Elección final: B4A

Como hemos expuesto, cada uno de los lenguajes propuestos en el apartado anterior tiene sus ventajas e inconvenientes, por lo que la decisión que tomemos no será objetiva. Todos ellos nos valdrían para llegar a la solución, todo depende de la importancia que le de cada usuario a las ventajas y desventajas. En nuestro caso, por ser usuario principiante y ante la recomendación del tutor de este proyecto, optamos por el Basic For Android. Esto nos permitirá empezar a programar en poco tiempo aplicaciones más complejas y resolver dudas en el fórum online a medida que vayan surgiendo. Esto es un punto muy importante. A menudo las dudas son resueltas por Erel, fundador de B4A, u otros usuarios con muchos más kilómetros de programación, lo que nos ayudará en sobremanera al desarrollo y solución de problemas de la aplicación.

4.2. BIBLIOTECA USB-CAN

Esta es la biblioteca en la que se centra nuestro proyecto. Mediante la infraestructura que tenemos, el USBtin establece comunicación entre un puerto USB y y un bus CAN. Está dividida en dos grandes regiones:

La primera región la componen las variables globales de la clase (Class_Globals) –en la programación orientada a objetos, una clase es una construcción que se utiliza para crear instancias de sí mismo - conocidas como instancias de clase, objetos de clase, objetos de instancia o simplemente objetos. Una clase define los miembros constituyentes que permiten a sus instancias tener estado y comportamiento. Los miembros de campo de datos (variables de miembro o variables de instancia) permiten que una instancia de clase mantenga el estado. Otros tipos de miembros, especialmente los métodos, permiten el comportamiento de las instancias de una clase. Las clases definen el tipo de sus instancias.

La segunda región está compuesta por las subrutinas o métodos desarrollados (Public Subs).

Ambas regiones están diseñadas para que sean compatibles con la biblioteca suministrada por el fabricante Kvaser (Kvaser es una empresa que proporciona soluciones avanzadas para bus CAN) .

```
'Class module
'CAN Library
'Hardware: USBtin by Thomas Fischl (Hardware version: 1.0; Firmware version: 1.05)
'Author: Marcel Garrido
'Data: 28/07/2015
'Version:1.0
'USBtin region based on "UsbSerial Ver 2.4" developed by JeanLC, B4A Community Member
'http://www.b4x.com/android/forum/threads/usbserial-library-2-0-supports-more-devices.28176/page-11#post-259167

Sub Class_Globals
    Dim usb1 As UsbSerial
    Dim astreams1 As AsyncStreams
    Dim state As Int = STATE_INI
    Dim inputChar As String
    Dim queueMSG As List
    Dim queueRPS As List

    Constants

End Sub

#Region Public Subs
'Initializes the object, the Message queue and the Response queue.
Sub Initialize
    queueRPS.Initialize()
    queueMSG.Initialize()
End Sub

USBtin

CANbus

#End Region
```

Figura 14 – USBCAN por regiones

4.2.1. Class Globals

El sistema de flujo de datos está basado en la conexión con el puerto USB, el flujo es asíncrono y las colas donde se almacenan los mensajes y respuestas entrantes. El puerto USB posibilita la comunicación con un elemento externo a nuestro Smartphone (USBtin). El flujo ha de ser asíncrono, ya que el tiempo de transmisión de datos es arbitrario, sin una frecuencia o periodicidad de ningún tipo y las colas se establecen diferenciadas, las que corresponden a una respuesta de una acción anterior (como podría ser un mensaje de confirmación a un comando previo) y los mensajes recibidos con información desde el controlador de convertidores (como podrían ser los parámetros actuales).

En esta región son definidas todas las variables globales y las constantes. Son dimensionadas:

- “usb1” como un UsbSerial, que permitirá hacer uso del microUsb de nuestro Smartphone.
- “astreams1” como flujo de información asíncrona.
- “queueMSG” y “queueRPS serán las dos colas que usarán en el sistema de escucha (Listener).
- “inputChar” es definido en este apartado también con el fin de ahorrarnos dimensionarlo cada vez que el Listener se activa.
- “state” es la variable que guarda el estado del autómata del Listener.

Por lo que respecta a las constantes, se han utilizado básicamente para mejorar la legibilidad del código y por limitar el número de inputs posibles en algunas funciones. Ejemplo:

```
'Baudrate
Dim Const BAUD_10K As Int = 0
Dim Const BAUD_20K As Int = 1
Dim Const BAUD_50K As Int = 2
Dim Const BAUD_100K As Int = 3
Dim Const BAUD_125K As Int = 4
Dim Const BAUD_250K As Int = 5
Dim Const BAUD_500K As Int = 6
Dim Const BAUD_800K As Int = 7
Dim Const BAUD_1M As Int = 8
'Can open modes
Dim Const can_NORMALMODE As Int = 1
Dim Const can_LOOPBACK As Int = 2
Dim Const can_LISTENONLY As Int = 3
'FSM states
Dim Const STATE_INI As Int = 0
Dim Const STATE_MSG As Int = 1
Dim Const STATE_RPS As Int = 2
```

4.2.2. Public Subs

Aquí se inicializan los objetos CANBUS, así como las dos colas. Está dividido en dos subregiones: USBtin y CANbus.

4.2.2.1. USBtin

Esta subregión establece el canal de comunicación a través del puerto microUsb con USBtin. Larespuesta de todos los mensajes enviados es tratada en el Listener de CANbus. Para la mayoría de los mensajes se utiliza la función *.getBytes("UTF8")* para pasar de una String –una *String* es una sucesión de caracteres ASCII almacenada en una cadena- a información en Bytes para así poder enviarla.

-UsbConnect

Comprueba si hay dispositivos conectados, demanda permisos de conexión, configura los divers y abre la conexión con USBtin con una velocidad determinada. La velocidad suportada en este caso es de 115200 bits/s. Inicializa

```
Sub usbConnect(baudrate As Int)
	If usb1.UsbPresent(1) = usb1.USB_NONE Then
		Log("Msgbox - no device")
		Return
	End If
	Log("Checking permission")
	If (usb1.HasPermission(1)) Then
		Dim dev As Int
		usb1.SetCustomDevice(usb1.DRIVER_CDCACM,0x4D8,0xA)
		dev = usb1.Open(baudrate, 1)
		Log(dev)
		If dev <> usb1.USB_NONE Then
			Log("Connected successfully!")

	astreams1.Initialize(usb1.GetInputStream,usb1.GetOutputStream,
"astreams1")
			Log("Initialized: " & astreams1.IsInitialized)
		Else
			Log("Error opening USB port 1")
		End If
	Else
		usb1.RequestPermission(1)
	End If
End Sub
```

-UsbClose

Corta el flujo de información así como la conexión con USBtin.

```
Sub usbClose
	astreams1.Close
	usb1.Close
	Log("Usb connection closed")
End Sub
```

-getHWversion

Envía un mensaje a la màquina USBtin pidiendo la versión del hardware. La respuesta es tratada por el Listener en CANbus. En condiciones normales, esta función devolverá la cadena “V1001”.

```
Sub getHWversion

	astreams1.Write((usb_VERSION_HW & Chr(CR) & Chr(LF)).GetBytes("UTF8"))

	Log("Got HWversion")

End Sub
```

-getFMversion

Envía un mensaje a la màquina USBtin pidiendo la versión del Firmware. La respuesta es tratada por el Listener en CANbus. En condiciones normales, esta función devolverá la cadena “V1005”.

```
Sub getFWversion
	astreams1.Write((usb_VERSION_FW& Chr(CR) & Chr(LF)).GetBytes("UTF8"))
	Log("Got FWversion")
End Sub
```

-UsbReadStatus

Envía un mensaje a USBtin preguntando por su estado. La respuesta, en condiciones normales, será la cadena “OK”.

```
Sub usbReadStatus
	astreams1.Write((usb_READSTATUS & Chr(CR) & Chr(LF)).GetBytes("UTF8"))
	Log("Status read")
End Sub
```

-usbSetTimestamping

Esta rutina envía un mensaje a USBtin activando o desactivando la impronta de tiempos (timestamping), marcando los datos de la comunicación con una fecha y hora concretes, para demostrar que una serie de datos ha existido y no han sido alterados desde un instante específico en el tiempo.

```
Sub USBSetTimestamping(bin As Int)
	astreams1.Write((usb_TIMESTAMPING&bin & Chr(CR)
&Chr(LF)) .GetBytes("UTF8"))
	If bin = 0 Then
		Log("Timestamping Off")
	Else If bin = 1 Then
		Log("Timestamping On")
	Else
		Log("Timestamping input error!")
	End If
End Sub
```

4.2.2.1. CANbus

Esta subregión gestiona la comunicación por canal CAN. Las primeras funciones que expondremos son de salida, y las cuatro últimas pertenecen al Listener asíncrono y el tratamiento de los datos obtenidos de la escucha.

-canOpen

Envía un mensaje a USBtin para abrir un canal de CAN. Según el modo esta subrutina sea llamada, se abre de manera estándar, en buble o en escucha.

```
Sub canOpen(mode As String)
	If mode = can_NORMALMODE Then
	astreams1.Write((can_OPEN_NORMALMODE&Chr(CR)&Chr(LF)).GetBytes("UTF8"))
		Log ("Open normal CAN channel")
	Else If mode = can_LOOPBACK Then

	astreams1.Write((can_OPEN_LOOPBACK&Chr(CR)&Chr(LF)).GetBytes("UTF8"))
		Log ("Open loop back CAN channel")
	Else If mode = can_LISTENONLY Then
	astreams1.Write((can_OPEN_LISTENONLY&Chr(CR)&Chr(LF)).GetBytes("UTF8"))
		Log ("Open listen only CAN channel")
	Else
		Log ("Error opening CAN channel. Only 1,2,3 are accepted as inputs")
	End If
End Sub
```

-canClose

Cierra un canal ya abierto del CAN.

```
Sub canClose
	astreams1.Write((can_CLOSE & Chr(CR) & Chr(LF)).GetBytes("UTF8"))
	Log ("Closed CAN channel")
End Sub
```

-canSetBitRate

Esta subrutina sirve para cambiar la velocidad de transmisión del canal CAN a una de las velocidades establecidas en el apartado de constantes (BAUD_10K, BAUD_1M).

```
Sub canSetBitrate(bitrate As Int)
	astreams1.Write((can_SETBITRATE&bitrate&Chr(CR)&Chr(LF)).GetBytes("UTF8"))
	Log("Baudrate changed")
End Sub
```

-canWrite

Envía un mensaje por el canal CAN. Como entrada hemos de establecer:

- El identificador del mensaje (id) como un número entero.
- La longitud del mensaje (dlc)
- El tipo de mensaje (mask)

Este último puede ser estándar, extendido, RTR estándar o bien RTR extendido. Se compara el valor de la máscara (mask) con las constantes predefinidas bit por bit y se obtienen las cuatro posibilidades, ya que la ausencia de un bit determina también posibles modos (por ejemplo, si no es estándar será extendido). Mediante las funciones IntsToBytes y HexFromBytes de la biblioteca ByteConverters se pasa de enteros a bytes y de bytes a hexadecimal. Sólo entonces se hace un casting a la cadena con la variable "flag" y se concatena el mensaje final. Esto se añade con el retorno de carro (CR) y el avance de línea, y se envía. Para evitar problemas con el índice, "val" ha de tener un valor de un array de dos ítems que serán enteros.

```
Sub canWrite(id As Int, msg() As Byte, dlc As Int, mask As String)
	Dim flag As String
	Dim bc As ByteConverter
	Dim val(1) As Int
	Dim d2 As String
	val(0) = id
	d2 = bc.HexFromBytes(bc.IntsToBytes((val)))
	If Bit.AND(mask,canMSG_RTR) = canMSG_RTR Then
		If Bit.AND(mask,canMSG_STD) = canMSG_STD Then
			flag = "r"&(d2.Substring(d2.length-3))
		Else
			flag = "R"&d2
		End If
	Else
		If Bit.AND(mask,canMSG_STD) = canMSG_STD Then
			flag = "t"&(d2.Substring(d2.length-3))
		Else
			flag = "T"&d2
		End If
	End If
	flag= flag & dlc
	For i = 0 To (dlc-1)
		val(0) = msg(i)
		d2 = bc.HexFromBytes(bc.IntsToBytes((val)))
		flag = flag & (d2.Substring(d2.length-3+1))
		Next
	astreams1.Write((flag & Chr(CR) & Chr(LF)).GetBytes("UTF8"))
	Log ("Message send: " & flag)
End Sub
```

-canSetAcceptanceFilterCode

Establece el filtro por Código del canal CAN. Solamente son relevantes los 11 primeros bits para USBtin. “Code” es una entra de tipo entero.

```
Sub canSetAcceptanceFilterCode(code As Int)
	astreams1.Write((can_FILTER_CODE&code&Chr(CR)&Chr(LF)).GetBytes("UTF8"))
End Sub
```

-canSetAcceptanceMaskCode

Establece el filtro por máscara del canal CAN. De nuevo solo los 11 primeros bits son relevantes. “Mask” es una entrada de tipo entero.

```
Sub canSetAcceptanceFilterMask(mask As Int)
	astreams1.Write((can_FILTER_MASK&mask&Chr(CR)&Chr(LF)).GetBytes("UTF8"))
End Sub
```

-Astreams1_NewData

Esta subrutina compone el Listener asíncrono en sí. El buffer almacena los datos que recibe por el microUSB de manera temporal. Para cada byte que obtiene recorriendo el buffer, activa la función FSM.

```
Sub Astreams1_NewData (Buffer() As Byte)
	For i = 0 To (Buffer.Length-1)
		FSM(Buffer(i))
	Next
End Sub
```

-FSM

Esta función es una máquina de estados finitos (Finite State Machine) que procesa los datos provenientes desde el Listener asíncrono, Astreams1_NewData. La variable de estado “state” contiene el valor de estado inicial, definido en las constantes de la Class

como STATE_INI, y comienza el proceso de reconocimiento del primer elemento de la cadena de entrada. En función del carácter inicial se modifica el estado a STATE_MSG en el caso que se trate de un mensaje de CAN (si comienza por “t”, “T”, “r” o “R”), o bien, STATE_RPS, en el caso de que sea una respuesta a una acción previa (como puede ser una comanda de la GUI). Entonces el autómata, a medida que procesa cada símbolo de la cadena proveniente del Listener, mantiene su estado hasta encontrar un carácter de finalización: retorno de carro (CR) o “7” (usb_ERROR). Después llena una de las dos colas, queueMSG o queueRPS, con toda la cadena recibida en el primer índice.

La variable “state” trabaja como una variable estática y almacena información del estado anterior. Dependiendo del estado actual y las condiciones de entrada se producen los cambios de estado y las acciones correspondientes. El estado inicial del autómata es único: STATE_INI, mientras que los estados finales pueden ser más de uno: STATE_MSG o STATE_RPS.

```
Sub FSM(data As Byte)
	Select state
		Case STATE_INI
			If data=CHR_cT OR data=CHR_t OR data=CHR_cR OR data=CHR_r Then
				inputChar = Chr(data)
				state = STATE_MSG
			Else
				inputChar = Chr(data)
				state = STATE_RPS
			End If
		Case STATE_MSG
			If data = CR Then
				queueMSG.add(inputChar)
				state=STATE_INI
			Else
				inputChar = inputChar & Chr(data)
			End If
		Case STATE_RPS
			If data = 7 Then
				queueRPS.Clear
				queueRPS.add(7)
				state=STATE_INI
			Else If data = CR Then
				queueRPS.add(inputChar)
				state=STATE_INI
			Else
				inputChar=inputChar & Chr(data)
			End If
	End Select
End Sub
```

-canReadResponse

Lee la cola de respuestas, queueRPS, generada por la función FSM y compara el primer elemento como una cadena con las constantes establecidas para identificar el mensaje. Si no hay respuesta disponible, la función devuelve inmediatamente el valor canERR_NOMSG. Interactúa y responde en consecuencia a las distintas situaciones que se puedan dar (determinadas por USBtin):

- Retorno de carro (CR): elemento de reconocimiento por excel·lència, se da en la mayoría de los casos con un funcionamiento correcto. Devuelve el valor "canERR_OK".
- Retorno de carro (CR) + "z": obtenido después de un mensaje enviado y recibido. Devuelve el valor "canERR_OK".
- "7": obtenido en caso que haya un error en el USBtin. Devuelve "canERR_KO".
- "F"+xx+CR: obtenido de la función canReadStatus. Xx son valores en hexadecimal que representan bytes conteniendo los siguientes valores de error:
    - Bit 1: señal de error
    - Bit 2: exceso de datos
    - Bit 5: error passivo
    - Bit 7: error de BUS

  Los bits no explicados no se usan. Devuelve xx.
- "Vxxxxx"/"vxxxxx": obtenido de la función getHWversion o getFMversion. Devuelve la versión requerida.
- Cualquier otro caso. Se devuelve canERR:KO, ya que los otros casos ya han sido testados

```
Sub canReadResponse() As String
	Log("queueRPS: "&queueRPS)
	If queueRPS.Size = 0 Then
		Log("error: "&canERR_NORPS)
		Return canERR_NORPS
	Else
		Dim first As String = queueRPS.Get(0)

		If first = usb_MSG_RECIEVED Then
			Log("MSG recieved")
			queueRPS.RemoveAt(0)
			Return canERR_OK
		Else If first = Chr(CR) Then
				queueRPS.RemoveAt(0)
				Return canERR_OK
		Else If first.Contains(usb_READSTATUS) Then
			Dim first As String = queueRPS.Get(0)
			Dim parsed As Int
			parsed=Bit.parseint(first.SubString(1),16)
			Log("Read status:"&parsed)
			queueRPS.RemoveAt(0)
			Return parsed
		Else If first.Contains(usb_VERSION_HW) Then
			Log("versio hw!")
			queueRPS.RemoveAt(0)
			Return first
		Else If first.Contains(usb_VERSION_FW) Then
			Log("versio fw!")
			queueRPS.RemoveAt(0)
			Return first
		Else If queueRPS.Get(0) = usb_ERR Then
			Log("USBtin error")
			queueRPS.RemoveAt(0)
			Return canERR_KO
		Else
			queueRPS.RemoveAt(0)
			Return canERR_KO
		End If
	End If
End Sub
```

-canReadMessage

Lee la cola de mensajes (queueMSG) generada por FSM y se compara el primer elemento como una String con las constantes establecidas para identificar el mensaje. Interactúa y responde en consecuencia a los posibles casos (estándar-“t”, extendido-“T”, RTR estándar “r” y RTR extendido “R”) y devuelve una lista en el que el modo figura como una string, el identificador como un entero, la longitud como un entero y la información del mensaje como una string. Si encontramos una respuesta desconocida, la función devolverá canERR_PARM. Si no hay respuesta, la función devolverá en este caso ERR_NOMSG. Una vez tratado se borra el primer elemento de la lista

```
Sub canReadMessage() As List
	Dim id, dlc As Int
	Dim mode, msg As String
	Dim list1 As List
	list1.Initialize
	If queueMSG.Size = 0 Then
		list1.Add(canERR_NOMSG)
		Return list1
	Else
		Dim first As String = queueMSG.Get(0)
		If first.Contains(Chr(CHR_cT)) Then
			mode = "Extended"
			id = first.SubString2(1,9)
			dlc = first.SubString2(9,10)
			msg=first.SubString(10)
			queueMSG.RemoveAt(0)
			list1.Add(mode)
			list1.Add(id)
			list1.Add(dlc)
			list1.Add(msg)
			Return list1
		Else If first.Contains(Chr(CHR_t)) Then
			mode = "Standard"
			id=Bit.parseint(first.SubString2(1,4),16)
			dlc = first.SubString2(4,5)
			msg=first.SubString(5)
			Log("Message recieved: "&queueMSG.Get(0))
			queueMSG.RemoveAt(0)
			list1.Add(mode)
			list1.Add(id)
			list1.Add(dlc)
			list1.Add(msg)
			Return list1
		Else If first.Contains(Chr(CHR_cR)) Then
			mode = "Extended RTR"
			id = first.SubString2(1,9)
			dlc = first.SubString(9)
			queueMSG.RemoveAt(0)
			list1.Add(mode)
			list1.Add(id)
			list1.Add(dlc)
			Return list1
		Else If first.Contains(Chr(CHR_r)) Then
			mode = "Standard RTR"
			id = first.SubString2(1,4)
			dlc = first.SubString(4)
			queueMSG.RemoveAt(0)
			list1.Add(mode)
			list1.Add(id)
			list1.Add(dlc)
			Return list1
		Else
			queueMSG.RemoveAt(0)
			list1.Add(canERR_PARAM)
			Return list1
		End If
	End If
End S
```

4.3 DESARROLLO DE LA APLICACIÓN

En este apartado realizaremos una visión por encima de la implementación de la aplicación y explicaremos su funcionamiento.

4.3.1. Primer problema: integrar Google Services a la Aplicación

Las especificaciones de la aplicación nos obligan a integrar los servicios de Google en nuestra aplicación para poder desarrollar las siguientes funcionalidades mediante las APIs que nos brinda Google en su página para programadores (365 días de uso gratis):

- Integrar un mapa y acoplarlo a nuestra aplicación. Para ello utilizaremos la API Google Maps.
- Acceder a los sitios que Google tiene almacenados en su base de datos. Esta tarea la realizaremos mediante la API Google Places. De esta manera podemos acceder a la base de datos de los restaurantes, hoteles, gasolineras y demás establecimientos, mostrarlos en el mapa y obtener información útil del sitio en cuestión.
- La misma API de Google Places tiene una función de búsqueda con autopredicción que nos será muy útil para la búsqueda de sitios por el usuario.

Los servicios de Google están diseñados para ser empleados en Android Studio, para integrarlos en Basic For Android es un poco más complicado. Lo explicaremos con el ejemplo de Google Maps:

El objetivo sería poder incluir un mapa en nuestra aplicación en una ventana de tamaño y localización modificables de la siguiente manera:

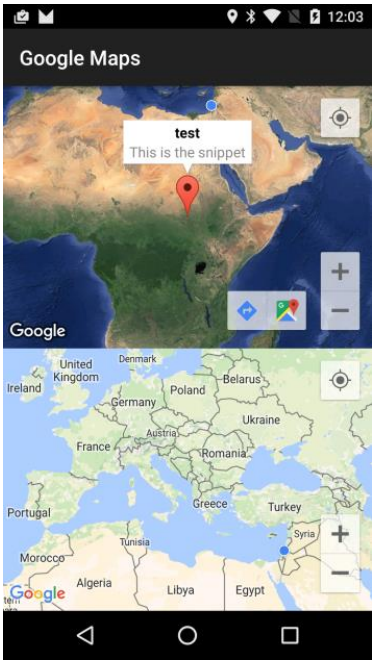


Figura 15.-Ejemplo GoogleMaps en B4A

Para ello seguimos los siguientes pasos (explicación obtenida del fórum de B4A):

- En primer lugar, debemos registrar nuestra app en la "developer console" de Google que se puede encontrar online en la siguiente dirección: https://console.developers.google.com/
- Una vez registrada la aplicación, necesitamos activar la API o APIs necesarias para esta aplicación. En este caso, activamos la API de Google Maps desde la misma developer console. Para ello clickaremos en Credentials->Create Credentials ->API Key->Android Key. Obtendremos una contraseña que deberemos guardar para usar en nuestra app.
- El siguiente paso sería incluir la librería o librerías necesarias en nuestra carpeta de librerías adicionales. Copiamos el archivo que contiene la librería GoogleMaps en nuestra carpeta. Las librerías se pueden obtener con las actualizaciones

disponibles en el Android SDK Manager que viene incluido en la descarga del B4A. Tan solo hay que moverlas a la carpeta de destino.

- En siguiente lugar debemos integrar los servicios de Google Firebase a nuestra aplicación. Para ello:
  - Desde el Android SDK Manager actualizamos e instalamos las últimas versiones de Android Support Repository y Google Repository (figura 16).

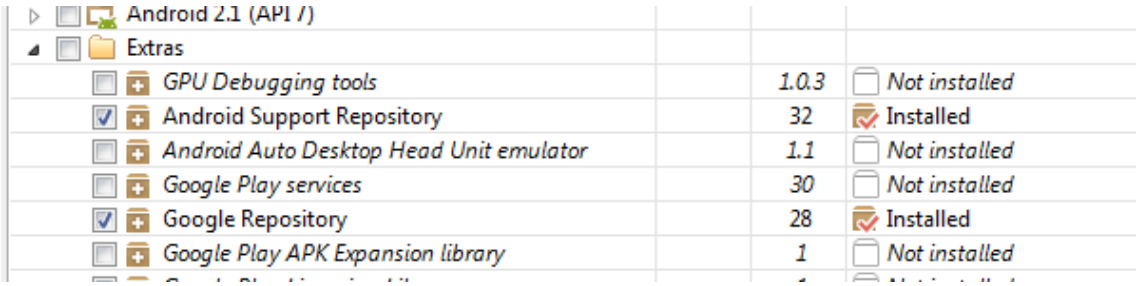


Figura 16

  - Una vez instalados, debemos registrar nuestra aplicación con Firebase y crear un nuevo proyecto (el "package name" de la aplicación debe coincidir)
  - Descargamos el archivo *google-services.json* que nos proveerá la página de Firebase y lo movemos a la carpeta contenedora del archivo b4a de nuestra aplicación.
  - Añadimos los "manifest snippets" que necesitamos a nuestro Manifest Editor en B4A, que en este caso sería:

```
Code:

'************ Google Play Services Base ************
AddApplicationText(
   <activity android:name="com.google.android.gms.common.api.GoogleApiActivity"
  android:theme="@android:style/Theme.Translucent.NoTitleBar"
  android:exported="false"/>
    <meta-data
  android:name="com.google.android.gms.version"
  android:value="@integer/google_play_services_version" />
)
'************ Google Play Services Base (end) ************
```

Figura 17

- Debemos añadir más líneas de código al Manifest Editor:

```
Code:

AddApplicationText(
<meta-data
  android:name="com.google.android.geo.API_KEY"
  android:value="AIzaSxxxxxxxxx "/>
)
```

Figura 18

Deberemos sustituir el Android:value por nuestra contraseña.

- Además, incluiremos en el código de nuestra aplicación la siguiente línea:

```
#AdditionalJar:com.google.android.gms:play-services-maps
```

Ya podemos usar los servicios de Google Maps en nuestra aplicación.

4.3.2. Pantalla principal

En esta pantalla incluiremos un fragmento de GoogleMap y botones para acceder a las distintas funciones de la aplicación de la siguiente manera:

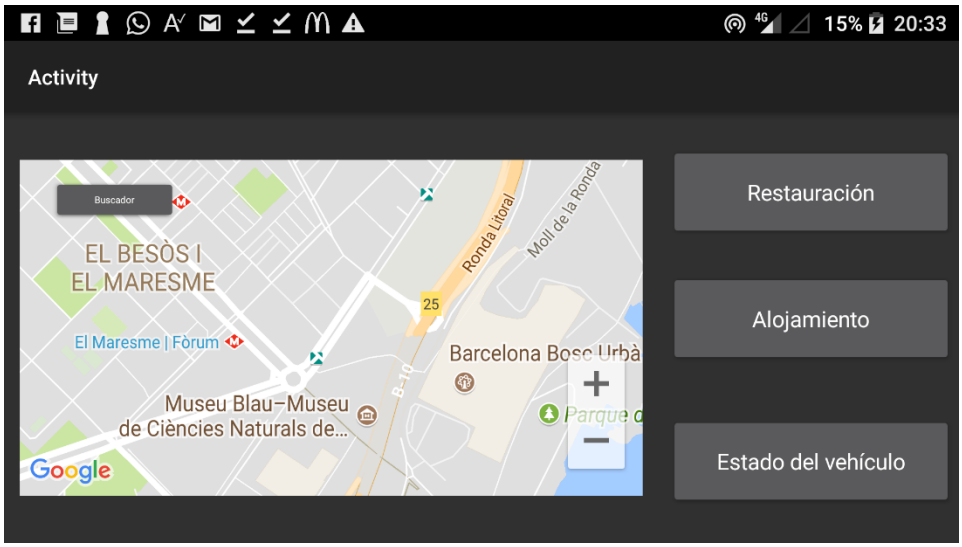


Figura 18.-Pantalla principal

Cada botón abre su correspondiente panel: el primero abre el panel que usaremos para desarrollar las funcionalidades de buscar sitio para parar a comer, el segundo el de las funciones que buscarán sitio para dormir y el tercero abrirá la ventana los estados leídos del vehículo por el bus CAN.

Asimismo, constará de un botón de búsqueda personalizada para que el usuario pueda buscar el destino que quiera (esquina superior izquierda mapa Figura18)

4.3.3. Panel 1

Cuando pulsemos el botón que abre este panel, la aplicación deberá buscar los establecimientos más cercanos para poder parar a comer. Los podrá mostrar en dos formatos: bien mediante una lista con información de algunos detalles de cada sitio o bien mediante Markers en el mapa.

Para desarrollar esta función nos basamos en las librerías de Google Places, que incluyen una herramienta de búsqueda con autopredicción de texto, y diferentes herramientas de búsqueda, como la que nos es más útil en este caso, un filtro por tipo de establecimiento con unas constantes establecidas en la base de datos de google que nos permitirá buscar solo los restaurantes cercanos.

Podemos encontrar toda la información acerca las funciones que nos proporciona Google en https://developers.google.com/android/reference/packages.

4.3.4. Panel 2

Este panel, como su propio nombre indica, nos ofrecerá los lugares más cercanos en los que el usuario pueda pararse a dormir. Funciona de manera similar al panel anterior, con el mismo interfaz que nos permite ver los posibles destinos en modo lista o en el mapa.

4.3.5. Panel 3

Este panel sí se diferencia de los anteriores. Se mostrarán en pantalla los niveles leídos provenientes del bus CAN en pantalla.

Constará asimismo de un botón para resolver incidencias, que nos mostrará directamente en el mapa el establecimiento más cercano para solucionar el problema. El mapa lleva integrado un botón que abre automáticamente la aplicación Navigator de Google para dirigirnos al destino.

Para desarrollar este apartado nos ayudamos de la biblioteca busCAN expuesta en la sección 4.2.Leemos los estados del vehículo mediante las funciones de lectura de esta biblioteca y las mostramos mediante unos gráficos en pantalla.

5.Impacto

Medioambiental

5. IMPACTO MEDIOAMBIENTAL

Este proyecto no destaca por un imapcto contaminante alto para nuestro ecosistema y por tanto no es necesario hacer un análisis exausto del impacto ambiental como en otros diseños para automóviles pesados.

No obstante se pueden considerar los siguientes aspectos a lo largo de la construcción del proyecto. Para que este proyecto sea possible ha sido necesario adquirir una tarjeta interfaz y un smartphone, los materiales y energías utilizadas para la producción y comercialización de estos objetos si puede generar un impacto ambiental. Ya que hay que tenir en cuenta que el telefóno móvil es uno de los aparatos más nocivos para el medioambiente de entre los que utilizamos en nuestro día a día. Uno solo puede contener hasta 40 materiales tóxicos entre los que destacan elementos químicos como el arsénico, antimonio, berilio, plomo, níquel y zinc. El coltán es normalmente el mineral que contiene estos elementos y es muy comun su uso con la finalidad de fabricar smartphones, debido a su bajo coste ya que se adquiere en zonas de conflcito por gente con escasos recursos ecónomicos. Este hecho ha sido denunciado por organizaciones y medios que demandan a las grandes compañías que busquen otras maneres para fabricar sus productos. El elemento más contaminante de un smatphone es la bateria, según los expertos, los agentes contaminantes de una bateria podrían contaminar 600.000 litros de agua por datos del Instituto Nacional de Estadística. Por tanto, hay que remarcar que una vez finalizado el uso de los dispositivos del proyecto, estos se deben desprender de manera correcta porque no se trabaja con materiales biodegradables o en el caso de que sea possible intentar reutilizar dichos materiales. Por otro lado, un smartphone de gama alta libera a la atmosfera 95 kilos de dióxido de carbono, como se señala en un estudio realizado por la Factultad de Ingeniería y Ciencias Físicas de la Universidad de Surey. Los resultados obtenidos apuntan a que la contaminación de un teléfono móvil està en gran medida relacionada, como hemos dicho anteriorment,con su producción, donde influyen los materiales usados, las energías o incluso el transporte

del producto hasta su punto de venta. También se puede considerar un pequeño consumo de energía eléctrica en el desarrollo de software.

En la fase de funcionamiento del proyecto se espera que este tenga un impacto positivo al mejorar la conducción de los vehículos, suponiendo cierto ahorro en el consumo de energía y una disminución de los contaminantes. El resultado de este proyecto permite implementar mecanismos para reduir el impacto ecológico debido a que da la posibilidad de encontrar las zonas de interés más próximas para reducir la distancia de conducción y todo el impacto ambiental que ello implica.

Conclusiones

CONCLUSIONES:

El aprendizaje de la programación nuestros días no es una tarea sencilla que estudies una vez y ya está. Los lenguajes de implementación están cambiando constantemente, desarrollados a cada y minuto y segundo por programadores de todo el mundo, y es por esto que uno debe evolucionar constantemente al ritmo de estos avances si no quiere quedarse atrás. No se trata de aprenderlo una vez y olvidarse, se debe tener una mente y abierta y consciente de este hecho, manteniéndose al tanto de los nuevos avances y tratando de aprender día a día. Lo importante es conocer cómo funcionan las estructuras estos lenguajes y saber manejarse no sólo en uno sino en varias para poder comprender mejor cómo funciona todo.

En un mundo en el que cada vez más cosas se encuentran presentes en “la nube”, el mundo del BIG DATA, los programadores tendrán un papel básico en esta sociedad donde el tratamiento de estas informaciones presentes en la red se presenta de una importancia clave para el crecimiento de las empresas si no quieren quedarse atrás.

Siendo realista, uno es consciente de que la aplicación desarrollada en este trabajo difícilmente tendrá salida en el mundo real. En mi opinión, este proyecto quedará tan solo como un trabajo con fines académicos. Esto es por varias razones, la primera es la limitación que nos ofrecen los fabricantes en sus lenguajes de comunicación con el bus CAN, pudiendo ofrecer este trabajo sólo a conductores de vehículo pesado. La segunda es que yo no soy un programador experto, este es mi primer trabajo y sé que es muy básico, pero mi intención es que sea un primer paso para poder desarrollar aplicaciones mucho más complejas en un futuro e iniciarme en un mundo tan complejo, a la vez que tan importante, como es el de programación para aplicaciones en un dispositivo tan necesario como es en nuestros días nuestro teléfono móvil.

Bibliografía

BIBLIOGRAFÍA.

-Javier Cuello y José Vittone, *Diseñando apps para móviles,2013.*

-M. Di Natale, “Understanding and using the Controller Area Network,” 2008.

-S. Corrigan, “Introduction to the Controller Area Network (CAN),” 2008

-https://petrolheadgarage.com/Posts/caracteristicas-de-un-sistema-can-bus/

-https://www.kvaser.com/

-http://standards.sae.org/j1939/71_201002/

-“Why many big IT consulting companies might soon become extinct?”:

http://sankeysolutions.com/why-many-big-it-consulting-companies-might-soon-become-extinct/

-“6 cosas que no te imaginarias que esconde una api en su funcionamiento” :https://bbvaopen4u.com/es/actualidad/6-cosas-que-no-te-imaginarias-que-esconde-una-api-en-su-funcionamiento

-“Futuro del automovil coche electric”: https://www.elconfidencial.com/motor/2017-01-04/futuro-del-automovil-coche-electrico_1311882/

-“Serial Control and Comunication VehicleWetwork” y “The SAe J1939 Comunications” en http://www.esd-electronics-usa.com/

- htp://www.microchip.com/stellent/idcplg? IdcService=SS_GET_PAGE&nodeId=1469 para “Introucción al bus CAN, sistemas embebidos”.